A Component Architecture for Platform-Independent Space Link Extension Services

Norman Lamarra and Imin Lin

Norman.Lamarra@jpl.nasa.gov
Imin.Lin@jpl.nasa.gov

Jet Propulsion Laboratory
California Institute of Technology
Pasadena , CA. 91109
October 1998

TABLE OF CONTENTS

TABLE OF FIGURES

Abstract

We are proposing a novel method of specifying and implementing CCSDS Space Link Extension (SLE) services. Traditionally, such services would be functionally specified (perhaps via an Application Programming Interface, or API) and accessed via a "wire protocol" implemented on a physical network (e.g., TCP/IP over Ethernet). Specification of such a service interface at the "wire level" typically is used to guarantee interoperability between separate implementations (e.g., between both sides of an interface agreement). However, when issues such as security are considered (some of which have not yet been adequately addressed), such a wire-level specification is likely to severely limit future implementations and capabilities of the SLE services. We are therefore proposing a novel layered architecture for the SLE services, which frees each party from having to interoperate at the wire level; effectively, we move the interface from the wire to the service with significant benefits. Our approach thus specifies the SLE services as a set of software components that communicate with each other using standard invocation methods implemented on every computer (e.g., "subroutine call"). Our definition of a component is: "A standalone implementation of an object interface which provides standard ways to find and invoke its methods". This "component approach" has many benefits, such as:

a) separates API specification from implementation issues (language, platform, etc.);
b) allows modular deployment of service components;
c) leverages recent advances in software development methodology, such as rapid prototyping and reusability;
d) facilitates operation over standards-based infrastructure;
e) frees application code from knowledge of lower layers;
f) allows applications and services to utilize modules dynamically.

We further propose to provide a layered implementation of these component interfaces, thus providing several options for a new kind of interface agreement:

1) one side could provide all the code implementing a specific wire protocol end to end;
2) each side could provide code implementing only its own wire protocol;

3) in each of the above cases, each side could selectively reuse modules from the other side;
4) in every case, the use of object-oriented methodology is recommended but optional.

Alternative (2) above thus allows each side to implement their own wire protocol (e.g., NASA could use plain TCP/IP, SSL over TCP, or DCE over TCP, while ESA could use plain TCP/IP, TP2, or CMIS/CMIP). In such a case, the two sides would meet at a designated software "gateway interface", which must be capable of communicating via both protocols. The proposed architecture dramatically simplifies the construction and maximizes the flexibility of such a gateway, and has many further benefits. For example: even if both sides of the gateway use the same underlying infrastructure (e.g., SSL over TCP), the proposed architecture still simplifies issues such as implementation and cross-management of the two security domains. It also allows each side to interoperate with third parties using yet another underlying infrastructure, thus allowing a single provider to implement several protocols simultaneously, with very little additional code for each protocol. Moreover, a provider can migrate its internal infrastructure (for example from plain TCP/IP to SSL), and hence its internal wire protocol, without affecting its interface to other users (i.e., by preserving the API).

1.0 Introduction and Background

Within NASA, the trend is from large multifunction missions to smaller, more focused missions that are "better, faster, cheaper". It is believed that this approach will improve the return on NASA's investment dollars, partly through increased opportunity to fund higher-risk smaller missions, but also through intelligent reuse of successful mission components. There are several keys to the success of this approach; for example, the ability to adapt rapidly to new mission requirements (perhaps including the use of new technologies). There is obviously less time to develop mission-specific architectures, and therefore a greater need for commonality (i.e., reuse) where possible. This approach is being taken in JPL's Advanced Deep Space Architecture (a.k.a. X2000), whose aims are to provide a common platform for flight and ground systems from which to specialize specific missions (in-situ, sample return, orbiter, etc.). A further issue being faced by NASA is that of outsourcing non-core activities, both to achieve the federally-mandated reduction in staffing and to leverage use of commercial and academic expertise.

Software development is clearly an area in which many of these issues have already been addressed (though not necessarily solved). Several new software development approaches have claimed to increase the adaptability, reusability, reliability, etc. of the software they produce, while reducing cost, time, and workpower required. Today's current wisdom claims that the use of Software Component Technology can provide many of these benefits most effectively. For this discussion, we define a software component to be: "a standalone implementation of an object interface which provides standard ways to find and invoke its methods". Use of component technology has mushroomed over the last few years in the commercial software industry, since it appears to be the most promising way to achieve the reductions in time to market and product adaptability required by the software marketplace in today's rapidly-changing economy.

In order to gain maximum benefit from the use of component technology, however, a software architecture is required which can effectively utilize components. Some goals and needs for such a software architecture are:

a) rapid adaptability (e.g., to new mission goals);
b) rapid assembly of new subsystems, mainly from available components;
c) a way to locate potentially usable components.

In turn, these needs require:

a) approaches to choose, design, build and manage components;
b) capability of demonstrating component use and measuring its benefits.

Issues involved in requirement (a) include: conducting domain analysis to identify the most useful components; building a framework for the design and implementation of the chosen

components; building component management capabilities for deployment and subsequent update of components. Issues involved in requirement (b) include: demonstration of usable components; evaluating the cost of producing and maintaining components for comparison to historical costs or those predicted if the component approach were not used.

1.1 TMOD Services

JPL's Telecommunications and Mission Operations Directorate (TMOD) provides a suite of Mission services, some which are grouped into several domains; for example; Telemetry, Command, and Data Management (TCDM); Multimission Image Processing (MIPS); and Common Services (CS). Beginning about two years ago, TMOD embarked on a software effort to achieve some of the above software goals in a specific subset of Deep-Space Network (DSN) software development activities. This activity has provided significant background for the SLE effort described here. The Software Reuse effort initially addressed the last-mentioned of the TMOD services (CS), since these were considered most likely to benefit from the component approach described above. Common Services include Monitor and Control, Network, Communications, and Data Delivery. The first components produced were produced from the Monitor and Control domain, and provided encapsulation of the TMOD MON-1 standards in first object, then component form. These components thus allow a new subsystem to be provided with generic monitor and control capabilities simply by utilization of the GIOMON1 API and dynamic linking with the GIOMON1 component. Specific examples of the benefit of such an approach are:

a) developers of the new subsystem do not have to deal with the underlying technology underlying the MON-1 protocol (which happens to be DCE-based);
b) developers of the new subsystem gain access to fully debugged and supported software which is maintained by another organization;
c) the object API of the GIOMON1 component allows significantly simpler architecture for publish and subscribe of monitor and control data from user applications, including a new capability to publish/subscribe objects.

1.2 Communication Services

TMOD Communication Services include the Fault-Tolerant Data Delivery (FTDD) service, which utilizes replicated servers to capture data published by producers and utilized by subscribers. This service is accessed by an API and implemented for a subsystem in the traditional way (i.e., procedural API and subroutine library linked with subsystem application code). This library and API are provided and maintained by TMOD-CS, who also maintain the MON-1 API and libraries.

These two examples of Common Services in current use exhibit some of the desired features mentioned above. For example, both provide encapsulation of their service in an API which hides the details of the underlying protocol (each of which is itself built upon standards

such as TCP/IP). However, there is an important difference between these two examples: the MON-1 API utilizes an underlying DCE-based protocol, which is developed using the DCE Interface Definition Language (IDL), while the FTDD service has no such IDL. The significance of this is that the IDL guarantees that the MON-1 protocol will work on any platform (regardless of implementation or language) which supports DCE. And although DCE itself uses TCP/IP or other networking standards, the actual MON-1 wire-level protocol (although completely specified by the IDL) is unimportant to the service implementers (barring performance considerations). Conversely, the FTDD service utilizes a proprietary wire protocol (itself using unicast or multicast UDP/IP) which must be specifically implemented and tested by the vendor separately on all participating platforms, even if they are know to support the UDP/IP standard. In this example, we believe that it requires much more work to maintain and port the latter protocol to new platforms or operating systems, since changes to the API may have different ramifications on each, while similar changes to the API via the former's IDL are guaranteed to be independent of platform, language, or operating system.

In this section, we have discussed some issues regarding communications protocols and some differences between specification at the wire level, the API level, and the IDL level. This is important background for the discussion below on the proposed SLE service architecture.

1.3 SLE Services

Initially, the authors were introduced to the RAF SLE service as an API specification written in DCE IDL and a protocol written in ASN.1; the former provides many benefits as outlined above, but requires the adoption of the DCE infrastructure, which may be undesirable for various reasons. The latter does not allow specification of an API, thus losing many of the advantages outlined above.

We therefore sought an approach that transcended the limitations of both these approaches, and achieving more of the desired goals mentioned in Section 1.0. While in the prototyping phase for the RAF service, we were also asked to consider CLTU service (required for several JPL missions) whose implementation schedule apparently preceded that for RAF. We therefore switched our attention primarily to CLTU, but wished to synthesize a consistent architecture for all the SLE services (RAF, CLTU, RVCF, FSP). In fact, most of the prototyping work we performed for the RAF service was directly applicable to the CLTU prototyping activity. We believe our approach achieves all of the above goals, as described in more detail below.

2.0 Component Software

Software languages have evolved over the past decades from early low-level languages (machine code, then assembly code) to higher-level languages such as FORTRAN and ALGOL in the 1960's, to C and Ada in the 1970's, to C++ and Smalltalk in the 1980's, and most recently to Java in the 1990's. Software methodology has similarly evolved from procedural to structured to object-oriented, and finally to component technology. This final stage represents

the first viable opportunity for real commercial success of the reusability promise, since this is the first technology to offer standalone implementation of required object-oriented functionality without the need for knowledge of implementation details such as language or platform. Recent but earlier attempts to provide some of these capabilities (e.g., DCE or CORBA) do not provide the full benefits of the component approach by themselves, though such infrastructure may be profitably used to implement platform-independent remote methods underneath the component architecture. In fact, Microsoft's® implementation of Distributed COM (DCOM) utilizes the DCE remote procedure call (RPC) to invoke remote methods (without requiring the rest of the DCE infrastructure). We begin by defining our component terminology.

2.1 Definition of a Component

For the purposes of this discussion, we define a component as: "a standalone implementation of an object interface that provides standard ways to find and invoke its methods". In turn, we define an object interface as "a cohesive set of methods that implement specific actions". (Neither definition needs to restrict the software to be object-oriented in general, though we believe the use of object-oriented technology significantly simplifies the implementation of the concepts described.) .

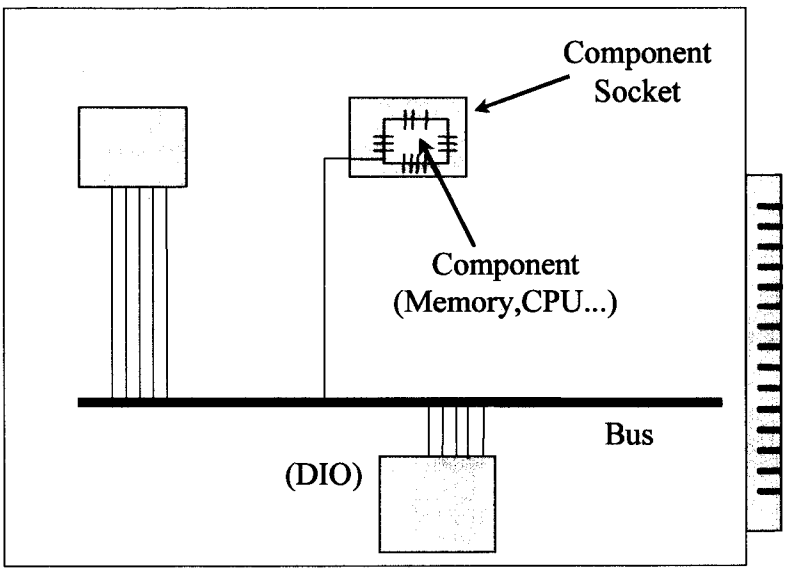

Figure 1: Analogy of Hardware Components

Figure 1 shows an analogy between software and hardware components. First, the "component socket" is analogous to the "object interface". Second, both software and hardware "components" perform a specific set of actions (such as read and write for a memory chip). Third, certain conventions assure the compatibility of equivalent components (e.g., pin compatible alternates; bus, power and ground pins, etc.).

2.2 Benefits of Component Approach

Successful use of the component approach to software development leads to the following benefits:

1) ***easy reuse***: software once developed and debugged can be used again in other contexts, facilitated by the component framework methodology (analogous to the hardware conventions described above);
2) ***easy assembly***: larger applications can be built from several pre-existing components, and functionality can be added or changed by replacement of one or more such component, again following similar rules to the "pin compatibility" analogy;
3) ***great flexibility***: such changes in functionality can, in fact, occur dynamically (i.e., at runtime), analogous to the hardware concept of "hot swappability";
4) use of components can ***enforce good object-oriented design***: it is more difficult to "cut corners" with component interfaces than with object interfaces, since the former must be capable of being utilized from many different environments, thus requiring more care in implementation; this benefit accrues only when the component is built as a specialized object structure;
5) ***cost savings***: many software vendors claim that their use of component technology significantly reduces cost and cycle time in all phases of the software lifecycle, namely design, development, integration and test, and maintenance; in some cases, it is claimed that the software development effort would not even be manageable without the use of components.

Unfortunately, these benefits do not come for free. First, there is a significant investment in training and experience required to succeed with components; this investment exceeds that required for good object-oriented practice, since components must be built even more carefully than equivalent objects. A particular example is the necessity for thread safety – since the component cannot know who will invoke its interfaces or how often, it is essential that every component be thread-safe; similarly, global variables cannot be used with components for the same reasons. In both examples, an object-oriented application could successfully violate these rules, though we would not recommend it.

2.3 C++ Construction of a Component

The C++ class structure with virtual inheritance provides a very good implementation template for the above component definition. In the example shown in Figure 2, we can define interfaces IF1 and IF2 each as a pure virtual class, but the class implementing the component can inherit from both interface classes. This indicates how a component can provide several interfaces, each with separate implementation of the same or different methods.

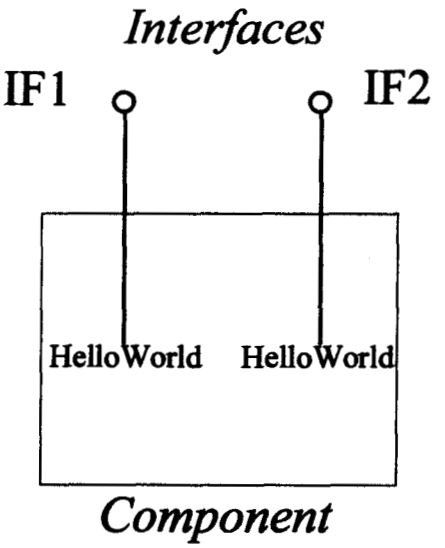

Figure 2: A Simple Example of a Component Carrying Two Interfaces

Example C++ code for this component is as follows:

```
interface IF1 : virtual public IUnknown
{
      virtual HelloWorld() = 0 ;
}

interface IF2 : virtual public IUnknown
{
      virtual HelloWorld() = 0 ;
}

class component : public IF1, public IF2
{
      HRESULT QueryInterface() ;
      unsigned long AddRef();
      unsigned long Release() ;
}
```

Note that the component contains only the three necessary, sufficient and mandatory methods for any COM component.

2.4 Examples of Components

A recent concept that has rapidly gained popularity is that of the "plug-in". For example, popular internet browsers provide an internal interface which allows new code to be added for the purpose of handling new data types which were unforeseen when the browser was initially installed (or even built). An example of such a plug-in is one to process streaming audio or video data. Such a plug-in fits the dynamic component concept well, since after the plug-in is installed, the browser utilizes the plug-in interface to route the appropriate data type to the plug-

in, which itself is dynamically linked and loaded only when needed (i.e., when such a datastream is recognized). This dramatically simplifies the deployment of the browser, since it needs only to provide and publicize the plug-in framework, and allows the implementation of specific plug-ins to be delegated to other vendors. Moreover, update or replacement of such plug-ins often does not need to affect the browser, since it often requires only replacement of a particular dynamic link library (DLL) on the browser machine's filesystem. Later in this paper, we will revisit this concept to show how our proposed SLE component architecture supports this elegant "plug-in" concept.

2.4.1 GIOMON1 Component

Before moving on to the SLE components, however, we present another example of both historical and relevant tutorial interest. As mentioned above, one of the existing TMOD Common Services is the MON-1 standard used for monitor and control of DSN subsystems. This service allows subsystems (such as the telemetry subsystem) to publish monitor data (such as health and status) and subscribe to control commands (such as configuration directives). It also allows the operator (sitting at a Network Monitor and Control (NMC) subsystem) to subscribe to such monitor data and publish such control commands. This set of services is implemented on top of a commercially-supplied DCE infrastructure (IBM/Transarc®), which provides a complete and integrated standard set of distributed services such as directory, file, time, and security, as well as robust automated service replication capabilities. Since all live subsystems in the DSN must be monitored and controlled, often by a single operator (and there are currently about 30 different subsystems), this service seemed an ideal candidate for the first TMOD component. The MON-1 API (specified via DCE IDL) was therefore encapsulated, first into an object API, then into a component (GIOMON1) carrying several interfaces, including ones for publish and subscribe, in a similar manner to that outlined for the HelloWorld component above.

Several benefits accrued from this encapsulation. For example, the underlying MON-1 service is not object oriented, thus it is not possible to "publish" an object, but only particular data types (integer, string, etc.). However, the encapsulated GIOMON1 service provides a simple mechanism to extend the capability of the MON-1 service to allow such objects to be published and subscribed. Many other extensions have been proposed (and some implemented); none has required any modification to the underlying MON-1 service.

Another example benefit of the GIOMON1 encapsulation is that it provides a simple way to allow a monitored subsystem to be relieved of the requirement to implement the DCE infrastructure (note that many of the 30 subsystems are relatively old, and some do not even support TCP/IP). Since the implementation details of the underlying MON-1 service are hidden from the user application via the GIOMON1 interface, the service can, in fact, exist on a different machine from the user application without its knowledge. This is an important precursor to the evolution of the SLE component architecture described below.

Appendix 9.1 gives sample code showing how a new subsystem could instantiate and use a publish/subscribe object by using the (dynamically linked) GIOMON1 component.

2.5 Platform Issues

Several component models exist today or are emerging: Java has the Beans component model, and CORBA is in the process of defining a component model. However, by far the most mature and widely deployed one is Microsoft's® Component Object Model. This developed initially in early Windows 3 as Dynamic Data Exchange (DDE), then evolved into Object Linking and Embedding (OLE), and later into Network OLE, ActiveX, OCX, DCOM, etc. For the purposes of making progress within TMOD in the component marketplace, we therefore chose the C++ language and the published COM specification ***without relying on any vendor implementation or tools***. The italicized words are emphasized to attenuate latent criticism directed at our selection of a particular language, vendor or proprietary component technology, but there is insufficient space here to develop the arguments and counter-arguments that have occurred. We merely note that the concepts described here could be implemented in other languages and component models (such as Java Beans) if required or sensible. We believe the most important point is that we are using open, published specifications for language and component model at the source code level, and thus we do not depend on any vendor licensing or hardware platform restrictions. Moreover, as shown in Figure 6 below, our component API can utilize code written in other languages (such as C or Ada), by wrapping it into our component shell to obtain the architectural benefits described (see component 3 with the square around it).

3.0 SLE Service Components

Having described the concept and benefits of a component architecture, and chosen an initial object model and language, we can now turn to the SLE service component architecture. The service interfaces can be naturally mapped into layers that will be described in further detail below. As we develop this layered description, we hope it becomes progressively clearer to the reader how naturally the component model fits the proposed service architecture.

3.1 Architectural Layers

Figure 3 shows the simplest possible partitioning of a service API for use by an application, while supplying the benefits described above (e.g., encapsulation). This picture fits both the MON-1 and FTDD APIs mentioned above. However, if the SLE service is implemented using components, then many benefits accrue. For example, different implementations of the service can be utilized (even dynamically) without impact to the user applications; this would not be achievable through the use of an API alone, but is enabled by the component approach. As a more tangible benefit, this approach also allows each party in an

interface agreement (e.g., JPL and ESOC) to implement and evolve the SLE services separately from their applications, while achieving interoperability both with the services and each other. It also provides some consistency in the architecture among the different services.

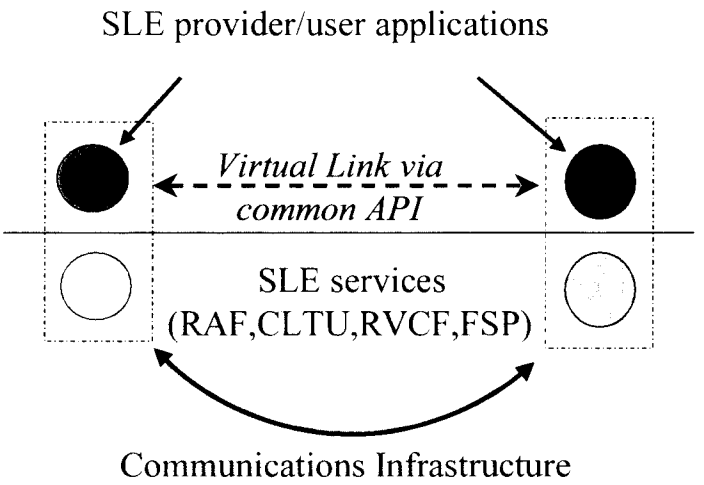

Figure 3: Service Overview

3.2 Common Application Programming Interface

We propose that a single object-oriented service API be carried throughout the chain from User application to Provider application, thus allowing each participant to directly utilize the SLE API while retaining the benefits of encapsulation of the implementation. This coupling is fully reversible, i.e., a different application could be swapped (even dynamically) on top of the SLE service in the same way. For example, User and Provider applications could be swapped.

Our view of the SLE service object in Figure 3 is thus that it represents the other end's application. For example, the SLE service provides a proxy of the Provider application to the User application. Or, more strikingly, the User application "plugs into" the Provider application (albeit indirectly). We have extended this concept to every layer of this communication as described below. This follows the concept of peer-to-peer virtual communication at every layer (cf. the ISO 7-layer stack), by using the component architecture to handle connections above and below each layer. Moreover, since the component architecture allows more than one interface to be carried by a component, as described in Section 2.3 above, we envision that the service component could carry one or more Administrative interfaces in addition to those for communication with the User or Provider above and with the Network below. In fact, even this Administration interface could be separated into "common" required methods (implemented by all parties to an interface agreement) and "private" optional methods (which are hidden from the other party and deal only with local issues such as the particular security model or the particular monitor and control mechanism).

3.3 Interface to Communications Infrastructure

Looking more closely at the service layers, we can now address the fact that the communications infrastructures may differ, depending on the domain. For example, JPL could choose DCE for data transport, while ESOC could choose TP2. Our approach takes care of such "mismatches" by implementing the same API for the appropriate domain at each layer.

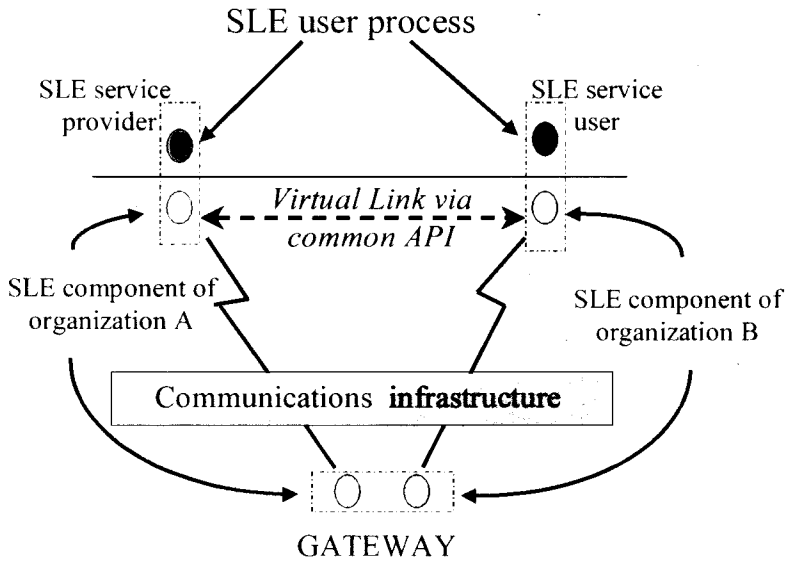

Figure 4: Service Component Architecture

Figure 4 shows a "gateway" representing the bridge between two such different communications infrastructures. Following our component architecture, such a gateway can be easily assembled from reusable components which will already have been built and tested by each side to implement their own lower layers. The issue of interoperation has now been localized to a single platform, thus avoiding language, operating system, or network mismatches. Next, we shall see how the component architecture facilitates even this interface.

3.4 Infrastructure Proxy

Looking from the Service Object's viewpoint, the application above and the network below are two users of its services (and carry the same API). In order to achieve this, we partition the service further into a "service only" component and a "domain proxy" component. Again, the proxy represents the "other end" to the service component. This is shown as the dashed line in Figure 5.

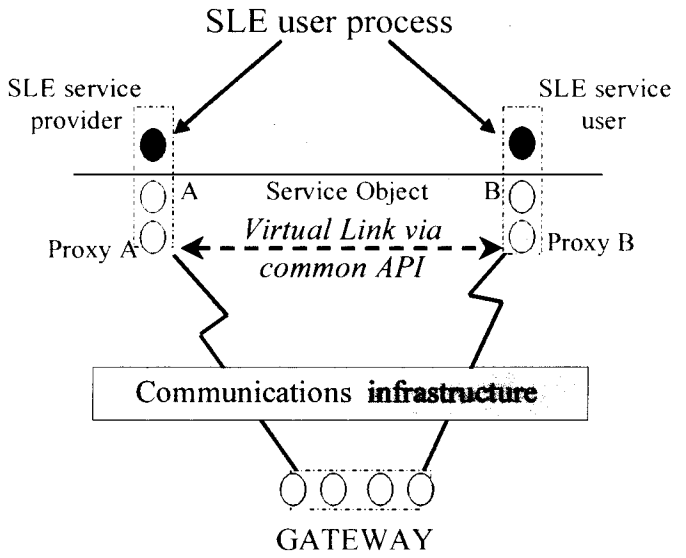

Figure 5: Reusable Implementation (via Proxies)

The obvious benefit of this approach (apart from joint ownership and reusability of the single API) is that several different domains can be served seamlessly via separate proxies (which contain and localize all domain-specific code but still conform to the single API). In the example JPL-ESOC chain mentioned in Section 3.3 above, all DCE-based information would be in a JPL's "DCE proxy", while all TP2-based information would be in ESOC's "TP2 proxy". This further frees the service object itself to be reused independently of the domain (Service Objects A and B could either be identical or different implementations, e.g., for platform, language, or political reasons).

3.5 Example Implementations

Example implementations of the "3-layer" approach described above are shown in Figures 6 and 7 for two situations:

1) Provider/User have the same network infrastructure;
2) Provider/User have different network infrastructures.

In Figure 6, the Provider Application is labeled 1, the User application 2, the Service Object 3, and the domain proxy 4. Note that this case does not require a gateway; the service on side B could be implemented in another language (perhaps not even object-oriented) provided it is wrapped into the service component API (as represented by the alternative path via a boxed version of service component 3).

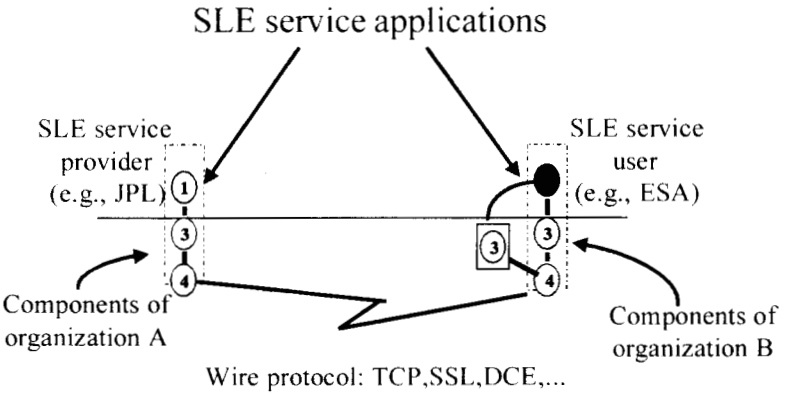

Figure 6: Example A: Provider/User have same network

Next, Figure 7 shows case 2, which requires only one more component: the second "domain proxy" (component number 5). The gateway, however, can now be assembled from components already produced by both sides.

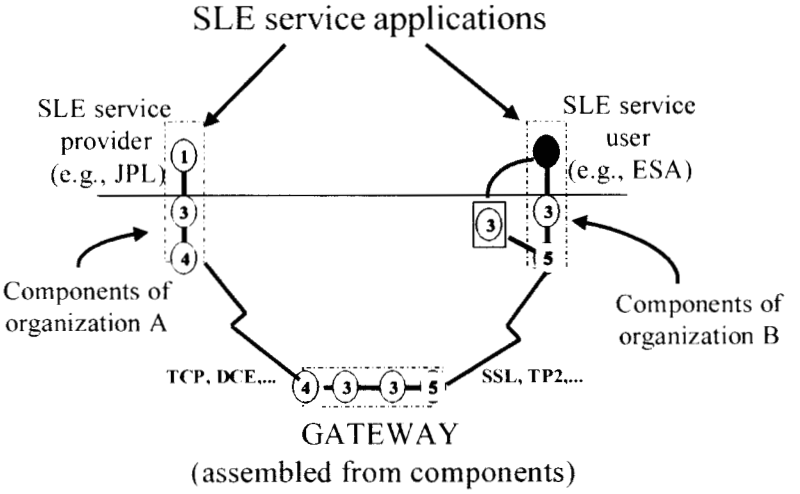

Figure 7: Example B: Provider/User have different networks

A key breakthrough here is that this configuration frees each side to independently specify (or even change) its internal wire protocol or infrastructure (including security domain). The other side doesn't even have to know what these are, as long as both sides agree to host that infrastructure on a common machine. This is because the "actual service interface" between the

two organizations occurs on this one machine (at the subroutine level between the two components numbered 3 inside the gateway below), each of which implements the same SLE API (and which could be interchanged). This therefore resolves all interface issues of differing platform, protocol, security domain, etc. by providing a common platform at which to meet, capable of hosting several protocols, security domains, etc. Moreover, the approach suggests that the software for this platform can be assembled from pieces already provided by each side. This approach thus allows greater independence, flexibility and much more simple implementation and management of a complex interface (e.g., spanning two security domains and network infrastructures) than that possible when the interface must be reduced to specifying bytes on a shared wire.

4.0 SLE Service Component Framework

In order for an application to utilize a service component in our architecture, an instance of the service object must be created (the example in Section 2.3 above showed an instantiation of the "HelloWorld" object via the component. Before the instance can be created, however, the appropriate component must be located and loaded. This is achieved by an infrastructure component that we call the Component Registration Component, or ComRegCom. This component has three tasks:

1) Component Registration: ComRegCom maintains a "component registration database" which provides component storage and query, i.e., a Component Repository.
2) Component Retrieving: ComRegCom could include authentication and authorization for allowable combinations of application and component.
3) Component Loading: ComRegCom loads the retrieved service component into the application process and returns the component interface to the application.
4) Component Release: ComRegCom destroys the service object and releases associated resources. (This step would be required if the service object were to be dynamically swapped by the user application).

This last step contains two substeps: first the loaded component exports the "component instantiation interface", and second, the application can find the desired interface by querying the loaded component, which responds by exporting the desired component interface.

4.1 Software Build Process

The above procedure utilizing ComRegCom is thus implemented in the following steps:

1 the application brings in the ComRegCom component through dynamic linking;
2 the application specifies the desired component and interface; ComRegCom starts to search for the component;
3 ComRegCom after finding the component brings it in through dynamic loading;

4 ComRegCom uses the Component Instantiation Interface to find the desired interface;
5 the interface is ready for use by the application.

4.2 Installation Process

For the specific case of the SLE service, ComRegCom can be reused to discover which components are needed for each situation (User, Provider, Gateway). This is achieved via an "SLE registration database" which contains information such as: number of components needed for each situation; type of components; valid instantiation values for each component; ordering information for the available configurations. The actual locations of the specified components are held separately in the "component registration database". For example, a particular mission might require communication between JPL and ESA via a gateway. The "SLE registration database" would contain the required information for the gateway for this mission, such as: which proxies are required for each side; which service components are required (RAF, etc.); instantiation values such as port ID number.

4.3 Configuration Management

The above architecture is highly flexible and provides reusability of key components. However, successful use of this architecture requires additional infrastructure to be provided. One of the important issues not addressed in this paper is that of Configuration Management (e.g., of component versions). In keeping with the distributed nature of the SLE services, such configuration management must necessarily be capable of being distributed if a single repository is not used for all pieces (e.g., a single shared database or a set of files in a globally-accessible distributed file system).

5.0 Conclusions

Our approach to defining a platform-independent architecture for SLE services using a common, component-based API achieves the following goals:

a) isolates API specification from implementation issues (language, platform, etc.);
b) allows modular deployment of service components;
c) leverages recent advances in software development methodology, such as rapid prototyping and reusability;
d) facilitates operation over standards-based infrastructures;
e) frees application code from knowledge of lower layers;
f) allows applications and services to utilize modules dynamically.

Further, our proposed layered implementation of these component interfaces provides several options for a new kind of interface agreement that does not require wire-level specifications to be agreed (or even exchanged). For example:

1) one side could provide all the code implementing a specific wire protocol end to end;
2) each side could provide code implementing only its own wire protocol;
3) in each of the above cases, each side could selectively reuse modules from the other side;
4) in every case, the use of object-oriented methodology is recommended but optional.
5) separate security policies can be implemented on each side

Alternative (2) above thus allows each side to implement their own wire protocol (e.g., JPL could use plain TCP/IP, SSL over TCP/IP, or DCE over TCP/IP, while ESOC could use plain TCP/IP, TP2, or CMIS/CMIP). In such a case, the two sides would meet at a designated software "gateway interface", which must be capable of hosting both protocols. The proposed architecture dramatically simplifies the construction and maximizes the flexibility of such a gateway, and has many further benefits. For example: even if both sides of the gateway use the same network infrastructure (e.g., SSL), the proposed architecture could still simplify interaction between the two security domains. It also allows each side to interoperate with third parties who may be using yet other underlying infrastructures. In fact, a single provider could implement several such protocols simultaneously, with very little additional code for each protocol (just the "domain proxy"). Moreover, a provider can migrate its internal infrastructure (for example from plain TCP/IP to SSL), and hence its internal wire protocol, without affecting its interface agreement with the other parties.

The component approach outlined above has been accepted for implementation by JPL and ESOC for the Integral and Cluster II missions. Within JPL, the component architecture is being implement for the SLE CLTU service (via new command and telemetry subsystems) as well as by independent contractors for the 26m antenna services (LEO missions). The gateway implementation team at JPL is responsible for integrating the required components and implementing the required databases (such as "SLE registration" and "component registration" databases). The matrix below shows example entries of implementation responsibility for particular SLE components.

	JPL gateway	JPL CMD	JPL TLM	ESOC	26m.
SLE application		uplink	downlink	up/downlink	up/downlink
SLE service component		CLTU	RAF, FSP	CLTU, RAF, FSP	CLTU, FSP
Socket proxy		CLTU			
DCE proxy	CLTU, FSP				
CMIS proxy				CLTU, RAF, FSP	
Infrastructure	ComRegCom, SLE reg db component reg db				

6.0 Acknowledgements

The work described was performed at JPL, California Institute of Technology under contract with NASA. The authors acknowledge the contribution of various software teams who contributed to the material described herein, or who are actually implementing subsystems to provide SLE services, reusable TMOD components, or both. This acknowledgement extends to the managers of these teams, working to test and verify the promise of Component Technology. This work was funded primarily by JPL's TMOD with additional support from JPL's Institutional Computer and Information Systems.

7.0 References

[1] Space Link Extension – Cross Support Concept Part 1. CCSDS 910.4-B-1. Blue Book. Issue 1. May 1996.

[2] Unpublished internal JPL reports on software reuse.

[3] Unpublished internal JPL description of TMOD service architecture

[4] "Component software: Beyond Object-Oriented Programming", Clemens Szyperski, ISBN 0-201-17888-5, 1998, Addison Wesley Longman Limited.

[5] "SOFTWARE REUSE: Architecture, Process and Organization for Business Success", by Ivar Jacobson,.Martin Griss. Patrik Jonsson. ISBN 201-92476-5, 1997. Addison Wesley Longman Limited

[6] "Inside COM: Microsoft's Component Object Model", by Dale Rogerson, ISBN 1-57231-349-8, 1997, Microsoft Press.

[7] Interface Control Document between ESA D/TOS and JPL TMOD for INTEGRAL, March 30, 1998 (contains draft SLE API).

8.0 Glossary of Acronyms

API	Application Programming Interface
CMIS	Common Management Infrastructure Service
CMIP	Common Management Infrastructure Protocol
COM	Microsoft® Common Object Model
DCE	Distributed Computing Environment (of The Open Group)
DSCC	Deep Space Communications Center (of JPL)
DSN	Deep Space Network (of JPL)
FTDD	Fault-Tolerant Data Delivery
GIOMON1	Generic Input Output – MON-1 component
IDL	Interface Definition Language
MON-1	TMOD's standard for Monitor and Control Services
SLE	Space Link Extension
SSL	Secure Socket Layer
TCP/IP	Transmission Control Protocol/Internet Protocol
TMOD	Telecommunications and Mission Operations Directorate (of JPL)
TP2	Transmission Protocol 2

9.0 Appendix: Code Examples

Section 9.1 illustrates the use of the GIOMON1 component described in Section 2.4.1 above.

9.1 Example Use of GIOMON1 Component

The code is shown in three parts:

a) the header file for the component interface, which contains definitions of the interface's methods and some constants (Section 9.1.1);
b) the header file for the component (Section 9.1.2), which contains the class definition and its method signatures;
c) and the actual code which instantiates the component (Section 9.1.3).

Note that this component contains two interfaces, just as the simple example in Section 2.3; the first interface is actually a "container" interface for the second, which is the actual component. The utility of this container interface is that this allows multiple instantiations of the publish/subscribe object to be handled uniquely, thus extending the functionality of the service as mentioned in Section 2.4.1. Moreover, it allows the user to know (via the reference counters) how many objects have been created; when these reference counters have returned to zero, then all objects have been released and the component can be dynamically unloaded. Without the container, this would not be so robust. This is important background for the SLE service component interface, which could also use this technique to handle multiple instantiations.

9.1.1 Header File for GIOMON1 Component Interface

```
/*************************************************************************
 *
 * Project: REUSE
 *
 * File name: GIOMON1IF.H
 *
 * ABSTRACT
 *
 *     This file defines the interfaces supported by the GIOMON1
 * component.
 *
 * DATE          NAME             REV      REMARKS
 *-----------------------------------------------------------------------
 *8/7/97        Imin Lin         0.0.1    Original Release
 *                                                                      *
 *************************************************************************/

#ifndef GIOMON1IF_H
#define GIOMON1IF_H

#include     "C_COM.H"
```

```
//IUnknown interface 822e8630-104f-11d1-80a1-00aa0027b016
#define IID_GIOMON1_COMPONENT_DEF
{0x822e8630,0x104f,0x11d1,{0x80,0xa1,0x00,0xaa,0x00,0x27,0xb0,0x16}}

//MON-1 CONTAINER interface 407862a0-0790-11d1-809e-00aa0027b016
#define IID_MON1CONTAINER_DEF
{0x407862a0,0x0790,0x11d1,{0x80,0x9e,0x00,0xaa,0x00,0x27,0xb0,0x16}}

//MON-1 PUBLISHER/SUBSCRIBER interface 470acc10-0790-11d1-809e-00aa0027b016
#define IID_PUBSUBER_DEF
{0x470acc10,0x0790,0x11d1,{0x80,0x9e,0x00,0xaa,0x00,0x27,0xb0,0x16}}

interface MON1_Container : IUnknown
{
   virtual void Container_setup(char* container_name)=0 ;
   virtual char* Container_who()=0 ;
   virtual ULONG Container_getrefct()=0 ;
} ;

interface MON1_Publish_Subscribe : IUnknown
{
   virtual void PubSub_setup(char* reg_name)=0 ;
   virtual char* PubSub_who()=0 ;
   virtual ULONG PubSub_getrefct()=0 ;
} ;

#endif
```

9.1.2 Header File for GIOMON1 Component

```
/*****************************************************************************
 *
 * Project: REUSE
 *
 * File name: GIOMON1.H
 *
 * ABSTRACT
 *
 *
 * DATE          NAME              REV      REMARKS
 *-----------------------------------------------------------------------
 *8/7/97        Imin Lin         0.0.1    Original Release
 *
 *****************************************************************************/

#ifndef GIOMON1_H
#define GIOMON1_H

#include    "C_COMIMPL.H"
#include    "GIOMON1IF.H"
#include    "C_string.H"

class GIOMON1 : public MON1_Container,
                public MON1_Publish_Subscribe
{
public:
    virtual HRESULT   __stdcall QueryInterface(const GUID& iid, void** ppv) ;
    virtual ULONG __stdcall AddRef()  ;
```

```
    virtual ULONG __stdcall Release() ;
    GIOMON1() ;

private:
    int current_IFidx ;
    IRefmg*  IRefmanager ;
    Cstring  container_nm ;
    Cstring  pubsub_nm ;

public:
    static double initflag ;
    static void init() { ; }
    virtual void Container_setup(char* container_name) {
        container_nm=container_name ; }
    virtual char*  Container_who() { return ((char*)(*container_nm)) ; }
    virtual ULONG Container_getrefct() { return IRefmanager->GetRefct() ; }
    virtual void PubSub_setup(char* reg_name) { pubsub_nm=reg_name ; }
    virtual char* PubSub_who() { return ("Pub Sub who") ; }
    virtual ULONG PubSub_getrefct() { return IRefmanager->GetRefct() ; }
} ;

#endif
```

9.1.3 Example Program Code Using the Component

After reviewing the component interface and class definition in the previous header files, the component user needs only to instantiate the component (via CoCreateInstance(), which can use the ComRegCom component described in Section 4.1 to locate, load, and link the component code), then use the component's interfaces directly. First, the code below creates a container object (mon1container) and proves the success by calling its "who()" method. Then it uses the standard QueryInterface() method to discover the pub/sub interface (which happens to be the second one as shown above), and proves the success by calling its "who()" method. After these few introductory calls, the main program can immediately start publishing and subscribing data.

```
#include    "C_COM.H"
#include    "C_COMreg.H"
#include    "GIOMON1IF.H"
#include    "C_string.H"
#include    <iostream.h>

IID IID_GIOMON1_COMPONENT=IID_GIOMON1_COMPONENT_DEF ;
IID IID_MON1CONTAINER=IID_MON1CONTAINER_DEF ;
IID IID_PUBSUBER=IID_PUBSUBER_DEF ;

main()
{
    MON1_Container* mon1container ;
    MON1_Container* mon1ref2 ;
    MON1_Publish_Subscribe* pubsub ;
    HRESULT result ;

    //Use the ComRegCom component to locate/link/load GIOMON1 component
    // and create a reference to the container object
    cout<<"About to create Component_Register==>"<<endl ;
    result=CoCreateInstance(IID_GIOMON1_COMPONENT,(IUnknown*)0,0,
```

```
                                IID_MON1CONTAINER,(void**)(&mon1container)) ;

    if(SUCCEEDED(result))
    {
        mon1container->Container_setup("MON container 1") ;
        mon1ref2=mon1container ;
        mon1container->AddRef() ;
        cout<<"Container - "<<mon1container>Container_who()<<"\n" ;
        cout<<"Referenced by -"<<mon1ref2->Container_getrefct()<<"\n" ;
    }
    else
    {
        cout <<"Container Interface failed\n" ;
        exit(0) ;
    }
    result=mon1ref2->QueryInterface(IID_PUBSUBER,(void**)(&pubsub)) ;
    if(SUCCEEDED(result))
    {
        pubsub->PubSub_setup("MON1 pub sub 1") ;
        cout<<"PubSub = "<<pubsub->PubSub_who()<<"\n" ;
        cout<<"Reference by -"<<pubsub->PubSub_getrefct()<<"\n" ;
        CoFreeUnusedLibraries(IID_GIOMON1_COMPONENT,IID_MON1CONTAINER) ;

    }
    else
        cout <<"PubSub Interface failed\n" ;
}
```